Efficient Pattern Matching in Elastic-Degenerate Texts*

Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis

Department of Informatics,
King's College London,
London WC2R 2LS, UK
{costas.iliopoulos, ritu.kundu, solon.pissis}@kcl.ac.uk

Abstract. Motivated by applications in bioinformatics, in what follows, we extend the notion of gapped strings to *elastic-degenerate strings*. An elastic-degenerate string can been seen as an ordered collection of solid (standard) strings interleaved by *elastic-degenerate symbols*; each such symbol corresponds to a set of two or more variable-length solid strings. In this article, we present an algorithm for solving the pattern matching problem with a solid pattern and an elastic-degenerate text running in $\mathcal{O}(N + \alpha\gamma mn)$ time; where m is the length of the pattern; n and N are the length and total size of the elastic-degenerate text, respectively; α and γ are parameters, respectively representing the maximum number of strings in any elastic-degenerate symbol of the text and the maximum number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text. The space used by the proposed algorithm is $\mathcal{O}(N)$.

Keywords: string processing algorithms, degenerate strings, indeterminate strings, elastic-degenerate strings, gapped strings

1 Introduction

Uncertainty in sequential data (strings) can be characterised using various representations. One such representation is a *degenerate string*, which is defined by the existence of one or more positions that are represented by sets of symbols from an alphabet Σ, unlike a solid (or deterministic, standard) string characterised by a single symbol at each position. For instance, $[\mathtt{a}\,\mathtt{b}]\mathtt{ac}[\mathtt{b}\,\mathtt{c}]\mathtt{a}[\mathtt{a}\,\mathtt{b}\,\mathtt{c}]$is a degenerate string of length 6 over $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$; and $\mathtt{abaababa}$ is a solid string of length 8 over $\Sigma = \{\mathtt{a},\mathtt{b}\}$. When a string is solid, we simply refer to it as string.

A *gapped string* is another way to capture uncertainty: it is an ordered collection of standard strings separated by variable-length gaps defined by an ordered collection of intervals [4]. Simply, a gapped string P can be represented as follows [15]: $P = P_1 *^{a_1,b_1} P_2 *^{a_2,b_2} P_3 \cdots *^{a_{\ell-1},b_{\ell-1}} P_\ell$, where $*$ is a *wildcard* symbol (also called *don't care* symbol or *hole*) that matches any symbol in alphabet Σ; $\forall i \in [1,\ell]$ each P_i is a string over Σ; and $\forall i \in [1,\ell-1]$ each pair

* This work was partially supported by the British Council funded INSPIRE Project.

(a_i, b_i) represents the gap (minimum and maximum number of wildcard symbols, respectively) between two consecutive strings P_i and P_{i+1}.

Here we introduce another representation to encapsulate uncertainty in sequential data—which we call *elastic-degenerate strings*—by extending and combining the ideas of gapped strings and degenerate strings. An *elastic-degenerate string* is a string such that an *elastic-degenerate symbol* can occur at one or more positions; each such symbol corresponds to a set of two or more variable-length strings. Another way to visualise an elastic-degenerate string is to see it as an ordered collection of $k > 1$ strings interleaved by $k - 1$ elastic-degenerate symbols. For instance, $\texttt{bc}\begin{bmatrix}\texttt{ab}\\ \texttt{aab}\\ \texttt{aca}\end{bmatrix}\texttt{ca}\begin{bmatrix}\texttt{abcab}\\ \texttt{cba}\end{bmatrix}\texttt{bb}$ is an example of an elastic-degenerate string over $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$.

This generalisation of the concept of *degeneracy* is motivated by several data mining problems [10] which can be reduced to the core task of discovering occurrences of one or more patterns in a text that can best be described as an ordered collection of strings interleaved by sets of variable-length strings.

More specifically, in genomics an important class of problems is to study within-species genetic variation; state-of-the-art solutions for this class comprises of matching (*mapping*) short strings (called *reads*) to a longer genomic sequence (canonical *reference genome* obtained through assembly). Owing to the high diversity among biologically relevant genomic regions in many organisms, the population level complexities cannot be captured by the linear structure of a reference genome (see [11]). Consequently, the recent research trend has shifted towards using alternative representations of genomic sequence for population-based genome assembly [8, 2, 5, 12]. One such representation that encodes a set of related genomes with variations in the reference genome itself (called Population Reference Genome in [12]), can be seen as an elastic-degenerate string.

The problem of pattern matching and discovery in the context of gapped strings has been studied extensively using combinatorial approaches (see [14] and references therein). However, a gapped string, which specifies the constraint on only the length of the gap between two consecutive strings P_i and P_{i+1}, differs from an elastic-degenerate string because the later precisely defines the possible strings (of varying lengths) that can exist between P_i and P_{i+1}. This precise identification of allowed strings in a gap makes the matching problem, in the context of elastic-degenerate strings, algorithmically more challenging.

In this article, we not only formalise the concept of elastic-degenerate strings but also present an efficient—in terms of both time and space—algorithm to solve the pattern matching problem in a given elastic-degenerate text. To the best of our knowledge, no other work, heretofore, explores the problem accounting for *elastic-degeneracy* in the text. In the next section, we introduce the basic definitions and establish the notions of elastic-degeneracy that will be used throughout. The algorithmic tools required to build the solution are described in Section 3. In Section 4, we formally define the problem along with presenting the algorithm. The algorithm is analysed in Section 5. Finally, the article is concluded in Section 6 with some remarks and future proposals.

2 Terminology and Technical Background

We begin with basic definitions and notation. We think of a *string* X of *length* $n = |X|$ as an array $X[1..n]$, where every $X[i]$, $1 \leq i \leq n$, is a *symbol* drawn from some fixed *alphabet* Σ of size $|\Sigma| = \mathcal{O}(1)$. The *empty string* of length 0 is denoted by ε. Σ^* denotes the set of all strings over alphabet Σ including the empty string ε. A string Y is a *factor* of a string X if there exist two strings U and V, such that $X = UYV$. We say that there is an *occurrence* of Y in X, or simply, that Y *occurs* in X, when Y is a factor of X. The starting position of an occurrence, say i, is called *head* of the occurrence and its ending position $i + |Y|$ - 1 is called *tail* of the occurrence. Note that an empty string occurs at each position in a given string. Consider the strings X, Y, U, and V, such that $X = UYV$. If $U = \varepsilon$, then Y is a *prefix* of X. If $V = \varepsilon$, then Y is a *suffix* of X.

A *degenerate symbol* $\tilde{\sigma}$ over an alphabet Σ is a non-empty subset of Σ, i.e., $\tilde{\sigma} \subseteq \Sigma$ and $\tilde{\sigma} \neq \emptyset$. $|\tilde{\sigma}|$ denotes the size of the set and we have $1 \leq |\tilde{\sigma}| \leq |\Sigma|$. A *degenerate string* is built over the potential $2^{|\Sigma|} - 1$ non-empty subsets of symbols of Σ. In other words, a degenerate string $\tilde{X} = \tilde{X}[1..n]$ is a string such that every $\tilde{X}[i]$ is a degenerate symbol, $1 \leq i \leq n$. If $|\tilde{x}[i]| = 1$, that is, $\tilde{X}[i]$ represents a single symbol of Σ, we say that $\tilde{X}[i]$ is a *solid* symbol and i is a *solid position*. Otherwise $\tilde{X}[i]$ and i are said to be a *non-solid symbol* and a *non-solid position*, respectively. For example, $[^{\texttt{a}}_{\texttt{b}}]\texttt{ac}[^{\texttt{b}}_{\texttt{c}}]\texttt{a}[^{\texttt{a}}_{\texttt{b}}]$ is a degenerate string of length 6 over $\Sigma = \{\texttt{a, b, c}\}$. A string consisting of only solid symbols is called a *solid string* or, simply, a string.

Now we give the terminology to build the concept of elastic-degeneracy by presenting the following definitions and examples.

Definition 1 (Seed: S). *A seed S is a (possibly empty) string over Σ.*

Definition 2 (Elastic-Degenerate Symbol: ξ). *An elastic-degenerate symbol ξ, over a given alphabet Σ, is a set of two or more strings over Σ (i.e. $\xi \subseteq \Sigma^*$ and $|\xi| > 1$). An elastic-degenerate symbol ξ is denoted by $\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_{|\xi|} \end{bmatrix}$, where each E_i, $1 \leq i \leq |\xi|$, is a solid string. The minimum (resp. maximum) length in ξ, denoted by $|\xi|_{min}$ (resp. $|\xi|_{max}$), is the length of the shortest (resp. longest) string in the set.*

Definition 3 (Elastic-Degenerate String: $\hat{X}$). *An elastic-degenerate string $\hat{X}$, over a given alphabet Σ, is a sequence $S_1\xi_1S_2\xi_2S_3 \ldots S_{k-1}\xi_{k-1}S_k$, where S_i, $1 \leq i \leq k$, is a seed and ξ_i, $1 \leq i \leq k-1$ is an elastic-degenerate symbol.*

An elastic degenerate string $\hat{X}$ can be visualised as follows:

$$\hat{X} = S_1 \begin{bmatrix} E_{1,1} \\ E_{1,2} \\ \vdots \\ E_{1,|\xi_1|} \end{bmatrix} S_2 \begin{bmatrix} E_{2,1} \\ E_{2,2} \\ \vdots \\ E_{2,|\xi_2|} \end{bmatrix} S_3 \ldots S_{k-1} \begin{bmatrix} E_{k-1,1} \\ E_{k-1,2} \\ \vdots \\ E_{k-1,|\xi_{k-1}|} \end{bmatrix} S_k.$$

Example 1 $\hat{X} = \texttt{abbc}\begin{bmatrix}\texttt{ab}\\ \texttt{aab}\\ \texttt{acca}\end{bmatrix}\texttt{cca}\begin{bmatrix}\texttt{aabcab}\\ \texttt{cba}\end{bmatrix}\texttt{bb}$ *is an elastic-degenerate string, where we have the following:*

- *Three seeds:* $S_1 =$ $\texttt{abbc}$, $S_2 = \texttt{cca}$, *and* $S_3 = \texttt{bb}$.
- *Two elastic-degenerate symbols:* $\xi_1 = \begin{bmatrix}\texttt{ab}\\ \texttt{aab}\\ \texttt{acca}\end{bmatrix}$ *and* $\xi_2 = \begin{bmatrix}\texttt{aabcab}\\ \texttt{cba}\end{bmatrix}$.
- *For* ξ_1*:* $E_{1,1} = \texttt{ab}$, $E_{1,2} = \texttt{aab}$, $E_{1,3} = \texttt{acca}$*; minimum length is* 2 *(length of* $E_{1,1}$*); and maximum length is* 4 *(length of* $E_{1,3}$*).*
- *For* ξ_2*:* $E_{2,1} = \texttt{aabcab}$, $E_{2,2} = \texttt{cba}$*; minimum length is* 3 *(length of* $E_{2,2}$*); and maximum length is* 6 *(length of* $E_{2,1}$*).*

Observe the use of $\hat{X}$ to distinguish an elastic-degenerate string from a solid string X or a degenerate string $\tilde{X}$. In the following, we define three characteristics of a given elastic-degenerate string $\hat{X}$ with k seeds.

Definition 4 (Total Size: $\|\hat{X}\|$). *The total size of* $\hat{X}$*, denoted by* $\|\hat{X}\|$*, is defined as the sum of the total length of its seeds and the total length of all the strings in each of its elastic-degenerate symbols:* $\|\hat{X}\| = \sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1}\sum_{j=1}^{|\xi_i|} |E_{i,j}|$.

Definition 5 (Length: $|\hat{X}|$). *The length of* $\hat{X}$*, denoted by* $|\hat{X}|$*, is defined as the sum of the total length of its seeds and the total number of its elastic-degenerate symbols:* $|\hat{X}| = \sum_{i=1}^{k} |S_i| + k - 1$.

Informally, the total number of positions in $\hat{X}$ is its length considering an elastic-degenerate symbol to occupy only one position. Intuitively, a position belonging to some seed will be called *solid position* and that of an elastic-degenerate symbol will be called *elastic-degenerate position.* In the running example, the total length of the seeds is 9; hence, $\|\hat{X}\| = 9+(2+3+4)+(6+3) = 27$, while $|\hat{X}| = 9 + 2 = 11$. The first `a` occurs at (solid) position 1, followed by `b` at (solid) position 2 and so on; $\xi_1$and ξ_2 are at (elastic-degenerate positions) 5 and 9, respectively; the last `b` is at (solid) position 11.

Definition 6 (Possibility-Set: $\Re$). *For the elastic-degenerate string* $\hat{X} = S_1\xi_1S_2\xi_2S_3\ldots S_{k-1}\xi_{k-1}S_k$*, its possibility-set* $\Re$ *is defined as*

$\Re = \{S_1E_{1,r_1}S_2E_{2,r_2}\ldots E_{k-1,r_{k-1}}S_k\}\ \ \forall r_i, 1 \leq i \leq k-1$ *such that* $1 \leq r_i \leq |\xi_i|$.

Informally, the *possibility-set* $\Re$ of $\hat{X}$ is the set of all possible solid strings obtained from $\hat{X}$. A solid string can be obtained by replacing each of the elastic-degenerate symbols with one of its constituent strings. In the running example, $\Re = \{$abbcabccaaabcabbb, abbcabccacbabb, abbcaabccaaabcabbb, abbaabccacbabb, abbcaccaccaaabcabbb, abbcaccaccacbabb$\}$. Note that constituent strings replacing the elastic-degenerate symbols have been underlined for clarity.

We are now in a position to define *matching* and *occurrence* in the context of elastic-degenerate strings.

Definition 7 (Matching). *An elastic-degenerate string $\hat{X}$ with k seeds and a solid string Y are said to match, denoted by $\hat{X} \simeq Y$, if, and only if, there exists a solid string $S = S_1 E_{1,r_1} S_2 E_{2,r_2} \dots E_{k-1,r_{k-1}} S_k, 1 \leq r_i \leq |\xi_i|$, obtained from $\hat{X}$ (i.e. $S \in \Re$ of $\hat{X}$), such that $S = UYV$, where $U, V \in \Sigma^*$, satisfying:*

$$\begin{cases} U = \varepsilon,\ V = \varepsilon & \text{if } S_1 \neq \varepsilon,\ S_k \neq \varepsilon \\ E_{1,r_1} \neq \varepsilon,\ V = \varepsilon,\ U \text{ is either empty or a prefix of } E_{1,r_1} & \text{if } S_1 = \varepsilon,\ S_k \neq \varepsilon \\ E_{k-1,r_{k-1}} \neq \varepsilon,\ U = \varepsilon,\ V \text{ is either empty or a suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 \neq \varepsilon,\ S_k = \varepsilon \\ E_{1,r_1} \neq \varepsilon,\ U \text{ is either empty or a prefix of } E_{1,r_1}, \\ E_{k-1,r_{k-1}} \neq \varepsilon,\ V \text{ is either empty or a suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 = \varepsilon,\ S_k = \varepsilon. \end{cases}$$

Informally, we say that $\hat{X}$ and Y match such that Y starts at the first position of $\hat{X}$ if the position is solid or as a suffix of one of its non-empty strings if it is elastic-degenerate; and Y ends at the last position of $\hat{X}$ if the position is solid or as a prefix of one of its non-empty strings if it is elastic-degenerate.

Example 2 *Consider $\hat{X}$ as given in Example 1. For string $Y =$ `abbcabccacbabb` we have that $\hat{X} \simeq Y$.*

Definition 8 (Occurrence). *In an elastic-degenerate string (text) $\hat{T}$, a solid string (pattern) P is said to have an occurrence starting and ending at positions i and j respectively, if $P \simeq \hat{T}[i\,..\,j]$. An occurrence is represented as the pair of starting position i (head) and ending position j (tail).*

For consistency with the intuitive meaning of an occurrence, we say that P occurs at the position of some elastic-degenerate symbol (say ξ_i) of $\hat{T}$, if it is a factor of any of the constituent strings of ξ_i.

Example 3 *Consider a pattern $P =$ `cabbcb` and a text $\hat{T}$ as follows:*

$$\texttt{aacabbcbbc}\begin{bmatrix}\texttt{a}\\ \texttt{aab}\\ \texttt{acca}\end{bmatrix}\texttt{bb}\begin{bmatrix}\texttt{c}\\ \texttt{acabbcbb}\\ \texttt{cba}\end{bmatrix}\texttt{bacabbc}\begin{bmatrix}\texttt{b}\\ \texttt{cabb}\\ \texttt{bbc}\\ \texttt{aacabb}\end{bmatrix}\texttt{cbc}.$$

All the occurrences of P in $\hat{T}$ are given below.

Occurrence:	$(3, 8)$	*(10,15)*	*(11,14)*	*(11,15)*	*(14,14)*	*(17,22)*	*(22,24)*
Strings chosen:	-	ξ_1: a ξ_2: c	ξ_1: acca ξ_2: cba	ξ_1: acca ξ_2: c	ξ_2: acabbcbb	ξ_3: b *or* ξ_3: bbc	ξ_3: cabb *or* ξ_3: aacabb

Note that more than one occurrence of P can start at the same starting position but their ending positions are different: for instance, $(11, 14)$ and $(11, 15)$ in Example 3. Also, note that different strings in the same elastic-degenerate symbols can lead to the same occurrence: for instance, the same pair of head and tail as happened for occurrences $(17, 22)$ and $(22, 24)$ in Example 3.

Example 4 *Here, we illustrate the case, where an elastic-degenerate string has the empty string as a seed. Consider a pattern* $P = \texttt{babbcb}$ *and a text* $\hat{T}$ *as follows:*

$$\hat{T} = \texttt{ab}\begin{bmatrix}\texttt{bcab}\\ \texttt{abb}\end{bmatrix}\begin{bmatrix}\texttt{ab}\\ \texttt{cbb}\\ \texttt{abc}\end{bmatrix}\texttt{cca}\begin{bmatrix}\texttt{bb}\\ \texttt{cb}\end{bmatrix}\texttt{ca}.$$

There is an occurrence of P *at* $(2, 4)$ *of* $\hat{T}$.

3 Algorithmic Tools

In this section, we briefly introduce a fundamental data structure, which supports a wide variety of string matching operations, and a well-known pattern matching algorithm. Both the data structure and the pattern matching algorithm will be used extensively by the proposed algorithm.

Suffix Tree

The *suffix tree* $\mathbb{S}(X)$ of a non-empty string X of length n is a compact trie representing all the suffixes of X such that $\mathbb{S}(X)$ has n leaves labelled from 1 to n. Additionally, each edge is labelled with a symbol of Σ. For any $i, 1 \leq i \leq n$, the concatenation of the edge labels on the path from the root of $\mathbb{S}(X)$ to leaf i is precisely the suffix $X[i\,..\,n]$. For any two suffixes $U = X[i\,..\,n]$ and $V = X[j\,..\,n]$ of X, if W is the *longest common prefix* (LCP) of U and V, then the path in $\mathbb{S}(X)$ corresponding to W is the same for U and V. In other words, the depth of the *least common ancestor* (LCA) of the two leaves is the same as the length of the LCP of the suffixes represented by those leaves. For a general introduction to suffix trees, see [3].

The construction of the suffix tree $\mathbb{S}(X)$ for string X of length n over a fixed-sized alphabet takes $\mathcal{O}(n)$ time and space using one of the algorithms in [18, 13, 17]. Once the suffix tree of X has been constructed, it can be used to support queries that return all Occ occurrences of a given string (called pattern) of length m in time $\mathcal{O}(m + Occ)$. In addition, the LCA of any two leaves of $\mathbb{S}(X)$, thus the length of the LCP of any two suffixes of X, can be computed in constant time after a linear-time pre-processing [7, 16]. A *generalised suffix tree* is a suffix tree for a set of strings [1, 6].

KMP Algorithm and Failure Function

Knuth, Morris, and Pratt (KMP) introduced a linear-time algorithm for pattern matching in [9]; that is, an algorithm for finding all occurrences of a pattern P in a text T. The KMP algorithm follows the naïve approach for this problem, that is, it slides P across T. Additionally, it pre-processes P by computing a *failure function* f that indicates the maximum possible shift using previously performed symbol comparisons. Specifically, the *failure function* $f(i)$ is defined as the length of the longest prefix of P that is a suffix of $P[1\,..\,i]$. By using the failure function, it achieves an optimal search time of $\mathcal{O}(n)$ after $\mathcal{O}(m)$-time pre-processing, where n is the length of T and $m < n$ is the length of P.

4 Algorithm for pattern matching in elastic-degenerate texts

4.1 Problem Definition

Problem. Pattern Matching in Elastic-Degenerate Texts
Input: *An elastic-degenerate text* $\hat{T} = S_1\xi_1S_2 \dots \xi_{k-1}S_k$ *of length* n *and total size* N*, a pattern* P *of length* $m < N$*.*
Output: *All the occurrences of* P *in* $\hat{T}$*.*

By definition, all the occurrences of the pattern P in the text $\hat{T}$ fall under the following cases:

1. P entirely lies in some seed.
2. P entirely lies in some string of an elastic-degenerate symbol.
3. P spans across one or more elastic-degenerate symbols. This can further be seen as:
   (a) P starts in some seed.
   (b) P starts in some string of an elastic-degenerate symbol.

For instance, consider Example 3: the occurrences $(3, 8)$ and $(14, 14)$ fall into Case 1 and Case 2, respectively; $(10, 15)$ and $(17, 22)$ fall into Case 3(a); $(11, 14)$, $(11, 15)$, and $(22, 24)$ fall into Case 3(b).

4.2 Algorithm

Note that a naïve solution to this problem would be to find the pattern occurrences in the possibility-set $\Re$ of $\hat{T}$ using the KMP algorithm; this time is exponential in the number of elastic-degenerate symbols. In this section, we present an efficient algorithm that makes use of the KMP algorithm and the suffix tree data structure. Clearly, the KMP algorithm can easily report the occurrences corresponding to the Cases 1 and 2. Case 3 requires some additional processing and data structures. The algorithm works in two stages, outlined below.

Stage 1: Pre-processing Pre-process the pattern P to compute its failure-function as required by the KMP algorithm. In addition, create the generalised suffix tree $\mathbb{S}_S$ for the set $\{P, S_1, S_2, \dots, S_k\}$ of strings corresponding to all the seeds of $\hat{T}$, as well as the generalised suffix tree $\mathbb{S}_\xi$ for the set $\{P\} \cup \xi_1 \cup \xi_2 \cup \dots \cup \xi_{k-1}$ of strings corresponding to all the strings in each of the elastic-degenerate symbols of $\hat{T}$. Furthermore, pre-process these two suffix trees so as to answer LCA queries in constant time.

Stage 2: Search Start searching the pattern P in the text $\hat{T}$ using the KMP algorithm, comparing the symbols and using the failure function to shift the pattern on a mismatch. The starting position of an occurrence being tested may be either solid or elastic-degenerate; we call the two types of occurrences as *Type* 1 and *Type* 2, respectively. We consider the two types separately as follows.

Type 1: Solid starting position Consider a situation when an occurrence starting from a position (say *pos*) that lies in some seed S_i is being tested. Proceed normally comparing the corresponding symbols of P and S_i; and shifting the pattern using failure function on a mismatch. As soon as the elastic-degenerate symbol ξ_i is encountered (suppose corresponding position in the pattern is p), abort the KMP algorithm (for this test). Check each of the strings of ξ_i (i.e. $E_{i,j}$) whether or not it occurs in the pattern at position p, using LCA queries on $\mathbb{S}_\xi$, and *tick* (mark) the tails of the found occurrences. This can be realised by maintaining a boolean array of size m, which we denote by $\mathbb{T}_i$.

Next, Procedure 1 (given formally below) is executed. Each ticked position of $\mathbb{T}_i$ is tried to extend by testing whether S_{i+1} occurs adjacent to it (using LCA queries on $\mathbb{S}_S$). For each such found occurrence of S_{i+1}, occurrences of strings of ξ_{i+1} are checked using the suffix tree $\mathbb{S}_\xi$ and their tails are ticked in $\mathbb{T}_{i+1}$. The procedure will then be repeated for $\mathbb{T}_{i+1}$; this continues recursively until there is no tail marked in some call.

Once the process ends (reporting all the occurrences of P starting from *pos*, if any), the failure function corresponding to the position where the KMP algorithm was aborted (i.e. p) is used to shift the pattern and the KMP algorithm resumes. It is to be noted that an occurrence of P is implied if the length returned by the LCA query between the pattern starting from some ticked-tail t and either of the following hits the boundary of the pattern:

- some seed S_i;
- any string $E_{i,j}$ of some elastic-degenerate symbol ξ_i.

Figure 1 elucidates the description given above.

Type 2: Elastic-Degenerate starting position Consider a situation when the starting position of an occurrence to be tested is an elastic-degenerate symbol ξ_i. This case can be processed in a similar fashion as the one described for Type 1, with the only difference being the manner in which tails are ticked initially.

Begin by applying the KMP algorithm for each $E_{i,j}$ to achieve two purposes: finding the occurrences of P in $E_{i,j}$ and ticking the last position of $E_{i,j}$ for which a prefix of P appears as a suffix of $E_{i,j}$. The ticked tails obtained in that way are then extended by Procedure 1 recursively and occurrences are reported. After the Procedure 1 ends, the KMP algorithm resumes and the testing starts at the beginning of the seed S_{i+1}.

5 Analysis

In this section, we discuss the correctness of the algorithm and analyse its space and time complexity.

Procedure 1: Procedure to extend ticked tails in a given $\mathbb{T}_i$ and reporting the occurrences found, if any.

```
Extend(T_i)
input  : A boolean array T_i of size m indicating ticked tails to be extended.
output : Reporting the found occurrences and preparing T_{i+1} for the next recursive call.
    isNonEmpty ← false;
    forall indices t of T_i which are ticked do
        l_s ← | LCA(P[t + 1 .. m], S_{i+1}[1 .. |S_{i+1}|]) |;
        if (l_s + t) == m then                                          // Pattern ends
            Report the occurrence;
        else if l_s = |S_{i+1}| then                                    // S_{i+1} occurs here
            e ← t + |S_{i+1}|;
            forall E_{i+1,j} in ξ_{i+1} do
                l_e ← | LCA(P[e + 1 .. m], E_{i+1,j}[1 .. |E_{i+1,j}|]) |;
                if (l_e + e) == m then                                  // Pattern ends
                    Report the occurrence (if not reported already);
                else if l_e = |E_{i+1,j}| then                          // E_{i+1,j} occurs here
                    Mark e + |E_{i+1,j}| − 1 in T_{i+1};
                    isNonEmpty ← true;
    if isNonEmpty then
        Extend(T_{i+1});
```

5.1 Correctness

Consider an occurrence (i, j). If the occurrence falls under the Case 1 (resp. Case 2) then $j = i + m - 1$ (resp. $j = i$) for some fixed i. Thus, the number of occurrences falling under either Case 1 or Case 2 is bounded by $\mathcal{O}(n)$. On the other hand, for occurrences under Case 3, let parameter γ represent the maximum number of elastic-degenerate symbols spanned by any occurrence (i, j). Note that γ captures the possibility that the elastic-degenerate symbols contain empty strings. As there can be maximum m prefixes going past an elastic-degenerate position, the number of occurrences per starting position i are bounded by $\mathcal{O}(\gamma m)$. Thus the total number of distinct occurrences (i, j) is bounded by $\mathcal{O}(\gamma mn)$.

The correctness of the presented algorithm is straightforward as every starting position of the text is being tested for potential occurrences exhaustively. While the occurrences corresponding to the Cases 1 and 3(a) are covered by Type 1, Type 2 investigates all occurrences associated with Case 2 and Case 3(b). Thus all the occurrences of P in $\hat{T}$ are reported.

5.2 Space Complexity

The space required by both, the failure-function and ticked tails array, is $\mathcal{O}(m)$. The suffix tree $\mathbb{S}_S$ uses $\mathcal{O}(m + \sum_{i=1}^{k} |S_i|)$ space and the suffix tree $\mathbb{S}_\xi$ uses $\mathcal{O}(m + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|)$ space. This leads to the total space required to be $\mathcal{O}(N)$, as $\sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}| = N$ and $m < N$.

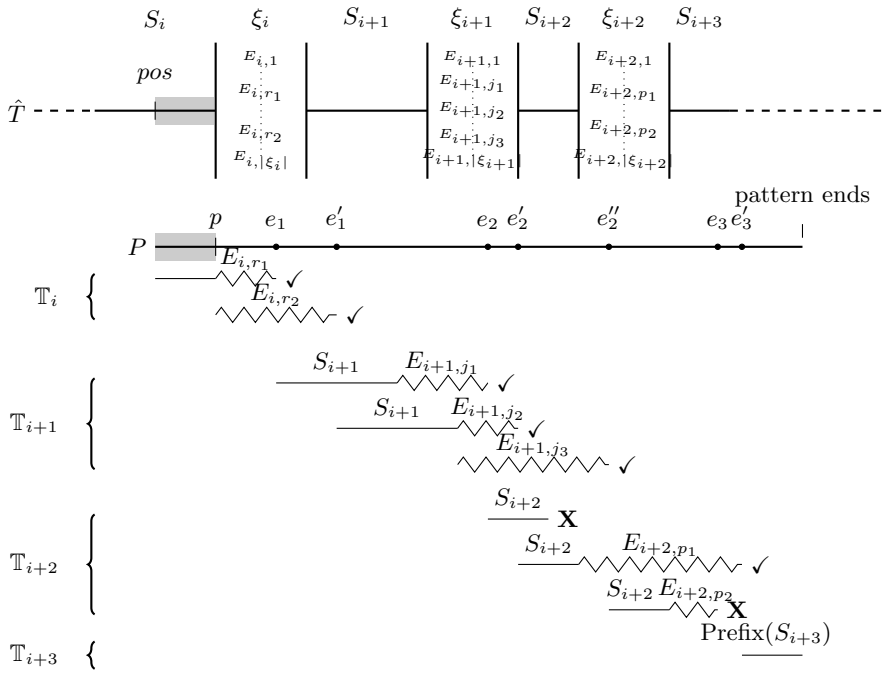

Fig. 1. An illustration of how the algorithm works for Type 1 occurrences. Strings in elastic-degenerate symbols are shown as zigzag, while solid lines depict the seeds. Symbol **X** denotes that this path could not be extended further while the symbol ✓ represents a ticked tail.

5.3 Time Complexity

The time taken by the pre-processing stage is $\mathcal{O}(N)$ as the failure function can be computed in $\mathcal{O}(m)$ time and construction of both the suffix trees (along with their pre-processing required to answer LCA queries in constant time) can be done in $\mathcal{O}(N)$ time.

The search stage uses the KMP algorithm over each seed and each string of every elastic-degenerate symbol in the text to report the occurrences for Case 1 and Case 2; and to search the beginning of the occurrence for Case 3. Thus the time consumed by the KMP algorithm is $\mathcal{O}(\sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|) = \mathcal{O}(N)$.

Procedure 1 can be analysed as follows. Intuitively, for every ticked position in the pattern (which can at most be m), an LCA query is used to find whether the corresponding seed occurs at the ticked position or not; a found such occurrence is then tried to extend by another LCA query with each of the strings in the following elastic-degenerate symbol. Let parameter α represent the maximum number of strings in any elastic-degenerate symbol of the text. This extension step for each ticked position will be carried out at most α times. More specifically,

the outer loop of the procedure runs m times and the inner one takes $\mathcal{O}(\alpha)$ time, as each LCA query takes constant time. Thus, each recursive call requires $\mathcal{O}(m\alpha)$ time. The number of recursive calls depends on the number of the elastic-degenerate symbols spanned by the occurrence of P being tested. In other words, if an occurrence spans across i elastic-degenerate symbols, there will be i recursive calls to the procedure. If γ is the maximum such i, Procedure 1 executes in $\mathcal{O}(\alpha\gamma m)$ time in total for each starting position.

Initial ticking of the tails in Type 1 needs $\mathcal{O}(\alpha)$ time. For Type 2, initial ticking is done by KMP algorithm (already accounted above). In the worst case, Procedure 1 will be called from each of the n starting positions of the text, leading to an overall time-complexity of the algorithm to be $\mathcal{O}(N + \alpha\gamma mn)$. In other words, the algorithm takes $\mathcal{O}(N + \alpha\gamma mn)$ time to find and report $\mathcal{O}(\gamma mn)$ number of possible occurrences of the pattern.

6 Final Remarks

Motivated by applications in bioinformatics, we extended the notion of gapped strings to elastic-degenerate strings. In particular, we presented an efficient algorithm for pattern matching in elastic-degenerate texts. Given a solid pattern and an elastic-degenerate text the presented algorithm runs in $\mathcal{O}(N + \alpha\gamma mn)$ time; where m is the length of the given pattern; n and N are the length and total size of the given elastic-degenerate text, respectively; α and γ are parameters, respectively representing the maximum number of strings in any elastic-degenerate symbol of the text and the maximum number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text.

Note that in applications involving gene sequence variation data, α represents the number of sequences in the multiple sequence alignment of the similar sequences and γ represents the number of genetic variation-sites falling in a full occurrence. The values of these parameters can be small and so the presented algorithm is expected to work very fast in practice. The space used by the algorithm is linear in the size of the input.

A proof-of-concept implementation of this algorithm (that has been tested for efficiency using synthetic data similar to the datasets used in genomics) can be accessed at `https://github.com/Ritu-Kundu/ElDeS`. Due to lack of space, experimental results are not included in the current version; they will be added in the full version of this article.

References

1. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. Journal of Computer and System Sciences 49(2), 208 – 222 (1994), `http://www.sciencedirect.com/science/article/pii/S0022000005800479`
2. Church, D.M., Schneider, V.A., Steinberg, K.M., Schatz, M.C., Quinlan, A.R., Chin, C.S., Kitts, P.A., Aken, B., Marth, G.T., Hoffman, M.M., Herrero, J., Mendoza, M.L.Z., Durbin, R., Flicek, P.: Extending reference assembly models. Genome Biology 16(1), 13 (2015), `http://dx.doi.org/10.1186/s13059-015-0587-3`

3. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007), 392 pages
4. Crochemore, M., Sagot, M.F.: Motifs in Sequences: Localization and Extraction, pp. 47–97. Marcel Dekker, New York (2004)
5. Dilthey, A., Cox, C., Iqbal, Z., Nelson, M.R., McVean, G.: Improved genome inference in the MHC using a population reference graph. Nat Genet 47(6), 682–688 (Jun 2015), http://dx.doi.org/10.1038/ng.3257, technical Report
6. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, NY, USA (1997)
7. Harel, H.T., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
8. Huang, L., Popic, V., Batzoglou, S.: Short read alignment with populations of genomes. Bioinformatics 29(13), i361–i370 (2013), http://bioinformatics.oxfordjournals.org/content/29/13/i361.abstract
9. Knuth, D.E., James H. Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977), http://dx.doi.org/10.1137/0206024
10. Li, Y., Bailey, J., Kulik, L., Pei, J.: Efficient matching of substrings in uncertain sequences. In: Zaki, M.J., Obradovic, Z., Tan, P., Banerjee, A., Kamath, C., Parthasarathy, S. (eds.) Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014. pp. 767–775. SIAM (2014), http://dx.doi.org/10.1137/1.9781611973440.88
11. Liu, Y., Koyutürk, M., Maxwell, S., Xiang, M., Veigl, M., Cooper, R.S., Tayo, B.O., Li, L., LaFramboise, T., Wang, Z., Zhu, X., Chance, M.R.: Discovery of common sequences absent in the human reference genome using pooled samples from next generation sequencing. BMC Genomics 15(1), 685 (2014), http://dx.doi.org/10.1186/1471-2164-15-685
12. Maciuca, S., del Ojo Elias, C., McVean, G., Iqbal, Z.: A Natural Encoding of Genetic Variation in a Burrows-Wheeler Transform to Enable Mapping and Genome Inference, vol. 9838, pp. 222–233. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-43681-4_18
13. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM (JACM) 23(2), 262–272 (1976)
14. Pissis, S.P.: MoTeX-II: structured MoTif eXtraction from large-scale datasets. BMC Bioinformatics 15(1), 235 (2014), http://dx.doi.org/10.1186/1471-2105-15-235
15. Rahman, M.S., Iliopoulos, C.S., Lee, I., Mohamed, M., Smyth, W.F.: Finding patterns with variable length gaps or don't cares. In: Computing and Combinatorics: 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006. Proceedings. vol. 4112, pp. 146–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11809678_17
16. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (Dec 1988), http://dx.doi.org/10.1137/0217079
17. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
18. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory. pp. 1–11. Institute of Electrical Electronics Engineer (1973)